Quick answers to common problems

Over 150 real-world recipes to simplify database management, automate repetitive tasks, and increase your productivity

SQL Server 2014 with PowerShell v5 Cookbook

PowerShell can be leveraged when automating and streamlining SQL Server tasks. This book starts with an introduction to the new features in SQL Server 2014 and PowerShell v5 and the installation of SQL Server. You will learn about basic SQL Server administration tasks and some security-related topics. You will explore different methods to back up and restore your databases and perform advanced administration tasks. The latter part of the book covers more advanced HADR tasks and techniques on developing scripts that work with BLOB, XML, and JSON.

Towards the end of the book, you will find additional resources, including a PowerShell tutorial, commonly-used PowerShell and SQL Server syntax, and a few online resources. There is also a mini-tutorial on how to create your own SQL Server sandbox Virtual Machine (VM).

Who this book is written for
If you are a SQL Server database professional (DBA, developer, or BI developer) who wants to use PowerShell to automate, integrate, and simplify database tasks, this book is for you. Prior scripting knowledge would be helpful, but it is not necessary.

What you will learn from this book

- Explore database objects and execute queries on multiple servers
- Manage and monitor the running of SQL Server services and accounts
- Back up and restore databases
- Create an inventory of database properties and server configuration settings
- Manage user permissions and security
- Work with CLR assemblies, XML, and BLOB objects in SQL
- Manage and deploy SSIS packages and SSRS reports

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Getting Started with SQL Server and PowerShell'
- A synopsis of the book’s content
- More information on SQL Server 2014 with Powershell v5 Cookbook
Donabel Santos (SQL Server MVP) is a business intelligence architect, trainer/instructor, consultant, author, and principal at QueryWorks Solutions (http://www.queryworks.ca/), based in Vancouver, Canada. She works primarily with SQL Server for database/data warehouse, reporting, and ETL solutions. She scripts and automates tasks with T-SQL and PowerShell and creates corporate dashboards and visualizations with Tableau and Power BI.

She is a Microsoft Certified Trainer (MCT) and an accredited Tableau trainer. She provides consulting and corporate training to clients and also conducts some of Tableau's fundamental and advanced classes in Canada. She is the lead instructor for SQL Server and Tableau (Visual Analytics) courses at British Columbia Institute of Technology (BCIT) Part-time Studies (PTS).

She is a self-professed data geek. Her idea of fun is working with data, SQL Server, PowerShell, and Tableau. She authored two books from Packt Publishing: SQL Server 2012 with PowerShell v3 Cookbook, and PowerShell and SQL Server Essentials. She has also contributed to PowerShell Deep Dives, Manning Publications. She blogs at http://sqlbelle.com/ and tweets at @sqlbelle.
PowerShell is a powerful and flexible task automation platform and scripting language from Microsoft. Many Microsoft applications, such as Windows Server, Microsoft Exchange, and Microsoft SharePoint now ship with PowerShell cmdlets that can be used for automated and streamlined integration. SQL Server database professionals can also leverage PowerShell to simplify database tasks using built-in cmdlets, the improved SQLPS module, the flexible SQL Server Management Objects (SMO), or by leveraging any of the readily available .NET classes.

SQL Server 2014 with PowerShell V5 Cookbook, provides easy-to-follow, practical examples for the busy database professional. There are over 150 recipes in this book, and you're guaranteed to find one that you can use right away!

You start off with basic topics to get you going with SQL Server and PowerShell scripts and progress into more advanced topics to help you manage and administer your SQL Server databases.

The first few chapters demonstrate how to work with SQL Server settings and objects, including exploring objects, creating databases, configuring server settings, and performing inventories. The book then dives deep into more administration topics such as backup and restore, managing security, and configuring AlwaysOn. Additional development and Business Intelligence (BI)-specific topics are also explored, including how to work with SQL Server Integration Services (SSIS), SQL Server Reporting Services (SSRS), and SQL Server Analysis Services (SSAS).

A short PowerShell primer is also provided as a supplement in the Appendix, which the database professional can use as a refresher or occasional reference material. Packed with more than 150 practical, ready-to-use scripts, SQL Server 2014 with PowerShell V5 Cookbook will be your go-to reference in automating and managing SQL Server.
What this book covers

Chapter 1, Getting Started with SQL Server and PowerShell, provides an introduction on how to work with SQL Server and PowerShell, including an introduction to SQL Server Management Objects (SMO). This chapter provides a recipe to install SQL Server using PowerShell and helps you explore and discover SQL Server-related objects and cmdlets.

Chapter 2, SQL Server and PowerShell Basic Tasks, provides scripts and snippets of code that accomplish some basic SQL Server tasks using PowerShell. Tasks include listing SQL Server instances, discovering SQL Server services, configuring SQL Server, importing/exporting records in SQL Server, and creating objects such as tables, indexes, stored procedures, and functions. Some recipes also teach you how to work with Azure SQL Database.

Chapter 3, Basic Administration, explores how administrative tasks can be accomplished in PowerShell. Some recipes deal with how to create SQL Server instances and database inventories, how to check disk space, running processes, and SQL Server jobs. Other recipes show you how to attach/detach/copy databases, add files to databases, and execute a query to multiple SQL Server instances.

Chapter 4, Security, focuses on how to work with SQL Server service accounts, manage logins/users/permissions, and monitor login attempts and also how to work with database roles, credentials, and proxies.

Chapter 5, Backup and Restore, teaches you what you already know about SQL Server backup and restore procedures and shows you how these tasks can be done using PowerShell. Many recipes use SQL Server-specific cmdlets, such as Backup-SqlDatabase and Restore-SqlDatabase wherever possible, but also utilize SQL Server Management Objects (SMO) to get more information on backup metadata. Some recipes also help you tackle backup and restore to Azure BLOB storage.

Chapter 6, Advanced Administration, discusses some of the most advanced features of SQL Server and how you can work with them in PowerShell. Recipes in this chapter include how to work with LocalDB, database snapshots, Filestream, FileTable, Full-Text Index, memory-optimized tables, security objects such as certificates, symmetric and asymmetric keys, and setting up Transparent Data Encryption (TDE).

Chapter 7, Audit and Policies, focuses on how to work with SQL Server tracking and auditing capabilities and SQL Server Policy Based Management (PBM). This chapter also explores how to work with SQL Server Profiler trace files and events programmatically.

Chapter 8, High Availability with AlwaysOn, covers specific recipes that can help you manage and automate SQL Server AlwaysOn, including how to install the failover cluster feature, enabling AlwaysOn, creating AlwaysOn availability groups and listeners, and testing the availability group failover.
Chapter 9, SQL Server Development, provides snippets and guidance on how you can work with XML, XSL, JSON, binary data, files in FileTable, and CLR assemblies with SQL Server and PowerShell.

Chapter 10, Business Intelligence, covers how PowerShell can help you automate and manage any BI-related tasks, including how to manage and execute SQL Server Integration Services (SSIS) packages, list and download SQL Server Reporting Services (SSRS) reports, and backup and restore SQL Server Analysis Services (SSAS) cubes.

Chapter 11, Helpful PowerShell Snippets, covers a variety of recipes that are not SQL Server-specific, but you may find them useful when working with SQL Server and PowerShell. Recipes include snippets for creating files that use timestamps, using Invoke-Expression, compressing files, reading event logs, embedding C# code, extracting data from a web service, and exporting a list of processes to CSV or XML.

Appendix A, PowerShell Primer, offers a brief primer on PowerShell fundamentals for the SQL Server professional. This chapter includes sections on how to run PowerShell scripts, understand PowerShell syntax, and convert scripts into functions to make them more reusable.

Appendix B, Creating a SQL Server VM, provides a step-by-step tutorial on how to create and configure the virtual machine that was used for this book.
Getting Started with SQL Server and PowerShell

In this chapter, we will cover:

- Working with the sample code
- Installing SQL Server using PowerShell
- Installing SQL Server Management Objects
- Loading SMO assemblies
- Exploring the SQL Server PowerShell hierarchy
- Discovering SQL-related cmdlets and modules
- Creating a SQL Server Instance Object
- Exploring SMO Server Objects

Introduction

If you have been working with Microsoft products, or have been managing or developing on the Microsoft platform, you might be familiar with PowerShell. If not, I can predict that you are bound to encounter it sooner than later. Since you are holding this book, my prediction just came true.
PowerShell is Microsoft's automation platform, which includes both a shell (often referred to as console) and a scripting language that allows one to streamline, integrate, and automate multiple tasks and applications. But why learn another technology and language? Why bother? If you have experience as a system administrator pre-PowerShell, you probably know the pain of trying to integrate heterogeneous systems using some kind of scripting.

Windows, at its core, is very Graphical User Interface (GUI)-driven with a lot of point-and-click, but this point-and-click is not so great when you have to do it several times over as you try to tie together systems and automate tasks.

Historically, the scripting solution would have involved a multitude of languages, including VBScript, Jscript, Perl, or Python; a batch file; and even a little bit of C, C++, or C#. System administrators had to be really creative and resourceful—duct taping a solution using a mishmash of languages in the absence of a real solution. It was messy, not flexible, and painful to maintain.

Enter PowerShell! PowerShell allows an administrator, or developer, to do more tasks in a faster, easier, and better way using a scripting language, now understood by many, in the Microsoft family of applications. PowerShell is now the one language you need to know if you want to automate and integrate either, within one application (for example, SQL Server), or between Microsoft and even non-Microsoft applications. Since many Microsoft products such as Windows Server, Exchange, SharePoint, and SQL Server have support for PowerShell, getting one system to talk to another is just a matter of discovering what cmdlets, functions, or modules need to be pulled into the script. The good thing is, even if the product does not have support for PowerShell yet, it most likely has .NET or .com support, which PowerShell can easily use.

PowerShell has become a major player in the automation and integration arena, and it will continue to be, for the foreseeable future.

**Working with SQL Server and PowerShell**

Before we dive into the recipes, let's go over a few important concepts and terminologies that will help you understand how SQL Server and PowerShell can work together.

**Running as an administrator**

Most of our recipes will perform possible queries and changes in your SQL Server instance or Windows Server. This will require elevated privileges both on your database side and in the PowerShell side. To ensure you can run the recipes in this book without getting access errors, you will need to execute the console or the ISE as administrator. One way to do this is by right-clicking on the PowerShell icon in your task bar and selecting to run either program as administrator.
You can confirm that you've launched either program as administrator by checking the title bar. You should see Administrator added to your title bar.

**Execution Policy**

The *Execution Policy* settings in PowerShell determine what is allowed or not allowed to be run in PowerShell.

For security reasons, PowerShell will not run automatically unless it is authorized in the settings. This is to prevent scripts from different sources, for example, the ones downloaded from the Internet, from potentially running malicious or destructive code.
To run the recipes in this book, you will need at least a RemoteSigned setting. To get this, run the following code:

```
Set-ExecutionPolicy RemoteSigned
```

### Running scripts

If you save your PowerShell code in a file, you need to ensure it has a .ps1 extension. Otherwise, PowerShell will not run it. Unlike traditional scripts, you cannot run a PowerShell script by double clicking the file. Instead, you can run this script from the PowerShell console simply by calling the name. For example, if you have a script called myscript.ps1 located in the C:\Scripts directory, you can provide the full path to the file to invoke it:

```
PS C:\> C:\Scripts\myscript.ps1
```

You can also change your directory to where the script is saved, and invoke it like this (notice there is a dot and backslash in front of the file):

```
PS C:\Scripts> .\myscript.ps1
```

If the file or path to the file has spaces, then you will need to enclose the full path and file name in single or double quotes. Before PowerShell v3, you would need to use the call (&) operator prior to the script name. From PowerShell v3 onwards, you do not need to specify the call operator anymore, but it will still work if you do:

```
PS C:\Scripts> & '.\my script.ps1'
```

If you want to retain the variables and functions included in the script in memory, so that it's available in your session globally, then you will need to dot source the script. Dot source means prepending the filename, or path to the file, with a dot and a space:

```
PS C:\Scripts> .\myscript.ps1
PS C:\Scripts> . '\my script.ps1'
```


### Running different PowerShell versions

If you are running PowerShell v3, v4, or v5, you can choose to run an older version of PowerShell. To do this, simply invoke the shell or start your session with a -Version parameter and provide the version you want to use:

```
Powershell.exe -Version 2
```
You can check that the change was made by using the $PSVersionTable variable. You should now see the PSVersion value reverted to the value you provided to the –Version parameter:

![PowerShell output showing $PSVersionTable](image)

**Line continuation**

Understanding how line continuation works in PowerShell will be crucial when working with the recipes in this book.

You will encounter a line of PowerShell code that may be wider than the width of the page you are reading. For example, consider the following code:

```powershell
# create your SQL Server SMO instance
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.Server -ArgumentList $instanceName
```

The preceding code, which creates a SQL Server instance, is meant to be written in a single line (no line breaks), otherwise it will not execute. However, to make the line more readable, some snippets may be broken into multiple lines as long as the line continuation character, a backtick (``), is provided at the end of the line just before the carriage return. Check out the following code:

```powershell
$server = New-Object ` -TypeName ` Microsoft.SqlServer.Management.Smo.Server ` -ArgumentList $instanceName
```

Adding the line breaks cleans up the code a little bit and makes it more readable. But do you see the backtick at the end of each line? You probably have to squint to see it. It's probably not obvious that the backtick is the last character before the carriage return.
The backtick (`) character in the U.S. keyboard is the key above the left Tab key, and to the left of the number 1 key. It shares the key with the tilde (~) sign. Check this post for the visual location of the backtick key in localized keyboards: http://superuser.com/questions/254076/how-do-i-type-the-tick-and-backtick-characters-on-windows.

For this book, I will try to avoid backticks for line continuation. Please assume that a long line of code in the recipes, although wrapping in your page, should be written as just a single line. You can also confirm the syntax by downloading the code from the Packt Publishing website.

Where possible, I will break down the code into more readable chunks without using the backtick. For example, consider a long line of code like this:

```powershell
$server.Databases | Get-Member -MemberType "Property" | Where-Object Definition -Like "*Smo*"
```

The preceding line can be rewritten into multiple lines using the pipe (|) operator at the end of each line:

```powershell
$server.Databases |
Get-Member -MemberType "Property" |
Where-Object Definition -Like "*Smo*"
```

If I have to use the backtick, I will call your attention to it in the code comments.

**PowerShell modules**

Modules are a way to extend PowerShell. Modules can add cmdlets and providers, and load functions, variables, aliases and other tools to your session.

For our recipes, we will use the SQLPS module a lot. To load this module, you can use the `Import-Module` cmdlet:

```powershell
Import-Module SQLPS
```

Note that running this command will change your current working directory to:

```powershell
PS SQLSERVER:>
```

**Working with the sample code**

This recipe simply walks you through how you can work with the scripts in this book. Samples in this book have been created and tested against SQL Server 2014 on Windows Server 2012 R2.
How to do it...

If you want to use your current machine without creating a separate VM, as illustrated in Create a SQL Server VM section in Appendix B, follow these steps to prepare your machine:


2. Microsoft provides really good documentation (both MSDN and TechNet) on how to install SQL Server, and the different ways you can install SQL Server. I encourage you to read the installation tutorial, as well as the installation notes that come with your software (https://msdn.microsoft.com/en-us/library/ms143219.aspx).

3. Install PowerShell v5. At the time of writing this book, only the Windows Management Framework (WMF) 5.0 Production Preview was available. You can download it from http://www.microsoft.com/en-us/download/details.aspx?id=48729. The installation instructions are also bundled in the download. At the time of writing this book, WMF 5.0 Production Preview can be installed on Windows 7 SP1, Windows 8.1 Pro/Enterprise, Windows Server 2008 R2, Windows Server 2012, and Windows Server 2012 R2.

4. By the time this book is in your hands, PowerShell v5 might be available and be bundled in the newer Microsoft operating systems. This download includes, as stated in the download page, updates to Windows PowerShell, Desired State Configuration (DSC) of Windows PowerShell and Windows PowerShell ISE. It also includes Package Management and Network Switch cmdlets.

5. If you are planning to use the console, set the execution policy to RemoteSigned in the console. This setting will enable us to run the scripts presented in this book.
   1. Right-click on Windows PowerShell on your taskbar and choose Run as Administrator.
   2. Set execution policy to RemoteSigned by executing the following on the console:

```
Set-ExecutionPolicy RemoteSigned
```

6. If you are planning to use the PowerShell Integrated Scripting Environment (ISE), set the execution policy to RemoteSigned. We will be using the improved ISE in many samples in this book.
   1. Right-click on Windows PowerShell on your taskbar and choose Run ISE as Administrator.
   2. Set the execution policy to RemoteSigned by executing the following on the script editor:

```
Set-ExecutionPolicy RemoteSigned
```
See also

- Keep up to date with PowerShell news, articles, tips, and tricks from the PowerShell team blog at http://blogs.msdn.com/b/powershell/.
- Check out the SQL Server PowerShell documentation on MSDN at https://msdn.microsoft.com/en-us/library/hh245198.aspx

Installing SQL Server using PowerShell

If you're really eager to dive into PowerShell and start installing SQL Server, this recipe will give you a taste of installing SQL Server with PowerShell using the SQL Server setup.exe file and a configuration file.

Getting ready

Get your SQL Server binaries ready. If you have it burned on a DVD, place your copy in the DVD drive. If you have it as an ISO or image file, mount the files now.

You will also need to identify the service accounts you want to use for the SQL Server services you want to install, as well as the locations for all the files that SQL Server will save on your system. In order to perform a completely automated install, the following script will need to be adjusted to use the default service account credentials, or specify the usernames and passwords within the $command variable.

In this exercise, we will generate a configuration (.ini) first, and then use this for the installation.

How to do it...

The steps to install a standalone SQL Server instance are as follows:

1. Generate the configuration file using the following steps:
   1. Load your SQL Server install disk or image and launch the setup.exe file.
   2. Go through the Wizard and enter all the configuration values.
   3. Once you get to the Ready to Install screen, note the Configuration file path:
4. Cancel the installation using the Wizard.

2. Change the configuration file to enable a silent install. Open the .ini file and make the following changes:
   1. Change the QUIET setting to True:
      ```
      QUIET="True"
      ```
   2. Comment out the UIMODE setting by putting a semicolon before it:
      ```
      ;UIMODE="Normal"
      ```
   3. Add the IAcceptSQLServerLicenseTerms value:
      ```
      IAcceptSQLServerLicenseTerms="True"
      ```
   4. Run your PowerShell ISE as administrator.
5. Add the following code to your script editor:

```powershell
# change this to the location of your configuration file
$configfile = "C:\Configurations\SQL_ConfigurationFile.ini"

# we are still using the setup.exe that comes with
# the SQL Server bits
# adjust the path below to where your setup.exe is
$command = "D:\setup.exe /ConfigurationFile={$configfile}"

# run the command
Invoke-Expression -Command $command
```

6. Change the location of the `$configfile` variable to the location where you saved your `.ini` file. Change the location of the executable as well. In the preceding script, the executable is in the `D:\` directory.

7. Execute the code.

### How it works...

SQL Server can be installed different ways:

- **Using Wizard:** You may choose to install SQL Server using the wizard-driven GUI approach, which starts by double-clicking on the `setup.exe` file that comes with your SQL Server binary (https://technet.microsoft.com/en-us/library/ms143219.aspx).

- **Via command prompt:** You can also install SQL Server using the command prompt by invoking `setup.exe` from the command prompt, and providing all the configuration values in the proper setup parameters (https://technet.microsoft.com/en-us/library/ms144259.aspx).

- **Via configuration file:** You can install SQL Server still by using the setup executable, but instead of providing all the values in the command prompt, use a configuration file that will host all the configuration values. Visit https://technet.microsoft.com/en-us/library/dd239405.aspx for more information.

- **Via SysPrep:** Install SQL Server using SysPrep. It is Microsoft's system preparation tool that allows administrators to deploy an image to multiple servers and/or workstations. Visit https://technet.microsoft.com/en-us/library/ee210664.aspx for more information.
In the recipe, we went with the third option and installed SQL Server using a configuration file. We are simply going to wrap a few components in PowerShell. You might be asking, "Why not script the whole process in PowerShell instead of using the executable and configuration file?" The answer is, we can do so, and there may be cases where that's the best approach. However, for simple and straightforward installations, it will be easiest to reuse as much of SQL Server's robust, tried-and-true installation process and wrap it inside PowerShell.

The SQL Server configuration file, which has the .ini extension, is a text file that contains installation parameter key-value pairs based on your entries and selections within the wizard. The format you will find in the file looks like this:

```plaintext
;comment or description
PARAMETERNAME = "value"
```

Some of the common parameters that will be specified in the configuration file include the following:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
<td>This is required to start the installation. It accepts only a single value of Install.</td>
</tr>
<tr>
<td>IACCEPTSQLSERVERLICENSETERMS</td>
<td>This is required for unattended installations, and it accepts End User License Agreement.</td>
</tr>
<tr>
<td>UPDATEENABLED</td>
<td>This specifies whether the installation should discover and include product updates, and it accepts True or False.</td>
</tr>
<tr>
<td>FEATURES</td>
<td>This specifies components to install, for example, SQLENGINE, AS, RS, IS, SSMS, or REPLICATION.</td>
</tr>
<tr>
<td>INSTANCENAME</td>
<td>This is a SQL Server instance name.</td>
</tr>
<tr>
<td>AGTSVCACCOUNT</td>
<td>This is a SQL Agent service account.</td>
</tr>
<tr>
<td>AGTSVCSTARTUPTYPE</td>
<td>This is a SQL Agent startup type, and it accepts any of the following values: Automatic, Manual, or Disabled.</td>
</tr>
<tr>
<td>SQLCOLLATION</td>
<td>This is the SQL Server instance collation.</td>
</tr>
<tr>
<td>SQLSVCCOUNT</td>
<td>This is the SQL Server database engine service account.</td>
</tr>
<tr>
<td>SQLSYSADMINACCOUNTS</td>
<td>These are the SQL Server system admin accounts.</td>
</tr>
<tr>
<td>TCPENABLED</td>
<td>This specifies whether an instance has TCP enabled.</td>
</tr>
</tbody>
</table>
Getting Started with SQL Server and PowerShell


You can create the .ini file from scratch, but it would be best to at least start with the configuration file you get with the wizard. From here, you can adjust and provide additional settings.

Once we've finalized the .ini file, the next step is to compose the actual command that needs to be executed. In the following code, we are simply creating a string that contains the path to the setup.exe and passing in a single parameter for the ConfigurationFile:

```powershell
$command = "D:\setup.exe /ConfigurationFile=$($configfile)"
```

Alternatively, you can also dynamically build the contents .ini file using PowerShell and then pass this configuration file to setup.exe, just like how we built $command previously.

Once the command string is ready, we can use the Invoke-Expression PowerShell cmdlet to run the expression contained by the $command variable:

```powershell
Invoke-Expression -Command $command
```

Instead of using the .ini file, you can also dynamically build all the parameters in a long string based on specific conditions or cases. You can take advantage of PowerShell's logic operators and other constructs when you do this. You should be able to compose the complete command and use Invoke-Expression to perform the actual installation:

```powershell
$command = 'D:\setup.exe /ACTION=Install /Q /INSTANCENAME="SQL01" /IACCEPTSQLSERVERLICENSETERMS /FEATURES=SQLENGINE,REPLICATION SQLSYSADMINACCOUNTS="QUERYWORKS\Administrator"'
```

**There's more...**

You can also take advantage of Desired State Configuration (DSC), which was introduced in PowerShell v4 and works with Windows Server 2012 R2, to install SQL Server.

DSC is a set of language extensions that will allow you to specify a desired state, or a set of ideal configurations, for your servers. This simplifies the configuration of new SQL Server instances, because all you have to do is to identify the desired state for your SQL Server installations and reuse the script for every deployment.

These are the simplified steps to take advantage of DSC:

1. Write a configuration script.
2. Run the configuration script to create a Management Object Framework (MOF).
3. Copy the MOF to the server you're installing SQL Server to. After the installation, at some point, you will want your server to pull the updated MOF automatically.
4. Apply the configuration to the target server and start the installation process.
The PowerShell team made the **xSqlPs** PowerShell module available, which is currently an experimental module, in the Technet Script Center (https://gallery.technet.microsoft.com/scriptcenter/xSqlps-PowerShell-Module-aed9426c). Here is a description of the xSqlPs module from the site:

*The xSqlPs module is a part of the Windows PowerShell Desired State Configuration resource kit, which is a collection of DSC resources produced by the PowerShell team. This module contains the xSqlServerInstall, xSqlHAService, xSqlHAEndpoint, xSqlHAGroup, and xWaitForSqlHAGroup resources.*

To install SQL Server, you will need to work with **xSqlServerInstall**. The PowerShell team has provided an excellent tutorial on how to use this DSC resource. This is a good starting script for a SQL Server Enterprise installation, and you can adjust it as needed. By the time this book is in your hands, the scripts in the module may have already been updated, or moved from an experimental to stable state. Please note that these scripts are also provided as is, with no support or warranty from Microsoft.

If you are looking for a good tutorial on DSC, check out the [Microsoft Virtual Academy](http://www.microsoftvirtualacademy.com/liveevents/getting-started-with-powershell-desired-state-configuration-dsc).

### Installing SQL Server Management Objects

**SQL Server Management Objects (SMO)** was introduced with SQL Server 2005 to allow SQL Server to be accessed and managed programmatically. SMO can be used in any .NET language, including C#, VB.NET, and PowerShell. Since SQL Server does not ship with many cmdlets, SMO is the key to automating most SQL Server tasks. SMO is also backwards compatible with previous versions of SQL Server, extending support all the way to SQL Server 2000.

SMO comprises two distinct classes: the **Instance** classes and the **Utility** classes.

The **Instance** classes are the SQL Server objects. Properties of objects such as the server, databases, and tables can be accessed and managed using the instance classes.

The **Utility** classes are helper or utility classes that accomplish common SQL Server tasks. These classes belong to one of four groups: Transfer, Backup, and Restore classes, or the Scripter class.

To gain access to the SMO libraries, SMO needs to be installed and the SQL Server-related assemblies need to be loaded.
Getting ready

There are a few ways to install SMO:

- If you are installing SQL Server 2014 or already have SQL Server 2014, SMO can be installed by installing Client Tools SDK. Get your install disk or image ready.
- If you want just SMO installed without installing SQL Server, download the SQL Server Feature 2014 Pack.

How to do it...

If you are installing SQL Server or already have SQL Server, perform the following steps:

1. Load up your SQL Server install disk or image, and launch the setup.exe file.
2. Select New SQL Server standalone installation or add features to an existing installation.
3. Choose your installation type and click on Next.
4. In the Feature Selection window, make sure you select Client Tools SDK.

5. Complete your installation

After this, you should already have all the binaries needed to use SMO.
If you are not installing SQL Server, you must install SMO using the SQL Server Feature Pack on the machine you are using SMO with. The steps are as follows:

2. Download the package.
3. Double-click on the SharedManagementObjects.msi to install.

There’s more...

By default, the SMO assemblies in SQL Server 2014 will be installed in `<SQL Server Install Directory>\120\SDK\Assemblies`. This is shown in the following screenshot:
Loading SMO assemblies

Before you can use the SMO library, the assemblies need to be loaded. With the introduction of the SQLPS module, this step is easier than ever.

Getting ready

In this recipe, we assume you have already installed SMO on your machine.

How to do it...

To load SMO assemblies via the SQLPS module, perform the following steps:

1. Open up your PowerShell console, PowerShell ISE, or your favorite PowerShell Editor.
2. Type the import-module command as follows:
   
   ```powershell
   Import-Module SQLPS
   ```
3. Confirm that the module is loaded by running the following. This should give the name of the module if it is loaded:
   
   ```powershell
   Get-Module
   ```

How it works...

The way to load SMO assemblies has changed between different versions of PowerShell and SQL Server. Before the SQLPS module and in PowerShell v1, loading assemblies could be done explicitly using the `Load()` or `LoadWithPartialName()` methods. The `LoadWithPartialName()` accepts the partial name of the assembly and loads from the application directory or the Global Assembly Cache (GAC):

   ```powershell
   [void][Reflection.Assembly]::LoadWithPartialName("Microsoft.SqlServer.Smo")
   ```

Although you may still see `LoadWithPartialName()` in some older scripts, this method is now obsolete and should not be used with any new development.

The method `Load()` requires the fully qualified name of the assembly:

   ```powershell
   [void][Reflection.Assembly]::Load("Microsoft.SqlServer.Smo, Version=9.0.242.0, Culture=neutral, PublicKeyToken=89845dcd8080cc91")
   ```
In PowerShell v2, assemblies can be added by using `Add-Type`:

```
Add-Type -AssemblyName "Microsoft.SqlServer.Smo"
```

When the SQLPS module was shipped with SQL Server 2012, loading these assemblies one by one became unnecessary, as long as the SQLPS module is loaded using the following code:

```
Import-Module SQLPS
```

There may be cases where you will still want to load specific DLL versions if you are dealing with specific SQL Server versions. Alternatively, you might want to load only specific assemblies without loading the whole SQLPS module. In this case, the `Add-Type` command is still the viable method of bringing the assemblies in.

**There's more...**

When you import the SQLPS module, you might see an error about conflicting or unapproved verbs:

```
WARNING: The names of some imported commands from the module SQLPS include unapproved verbs that might make them less discoverable. To find the commands with unapproved verbs, run the Import-Module command again with the Verbose parameter.
For a list of approved verbs, type Get-Verb.
```

This means there are some cmdlets that do not conform to the PowerShell naming convention, but the module and its containing cmdlets are still all loaded into your host. To suppress this warning, import the SQLPS module with the `-DisableNameChecking` parameter.


**See also**

- The *Installing SQL Server Management Objects* recipe.
Getting Started with SQL Server and PowerShell

Exploring the SQL Server PowerShell hierarchy

SQL Server started shipping with the SQLPS module in SQL Server 2012. The SQLPS module allows PowerShell to access SQL Server-specific cmdlets and functions, and also loads commonly used assemblies when working with SQL Server. This continues to be the case in SQL Server 2014.

Launching PowerShell from SQL Server Management Studio (SSMS) launches a Windows PowerShell session which imports the SQLPS module automatically, and sets the current context to the item the PowerShell session was launched from. Database administrators and developers can then start navigating the object hierarchy from there.

Getting ready

To follow this recipe, you should log in to SQL Server 2014 Management Studio.

How to do it...

In this recipe, we will navigate the SQL Server PowerShell hierarchy by launching a PowerShell session from SQL Server Management Studio:

1. Right-click on your instance node.
2. Click on **Start PowerShell**.
Note that this will launch a PowerShell session and load the SQLPS module. This window looks similar to a command prompt, with a prompt set to the SQL Server object you launched this window from. In the following screenshot, **ROGUE** refers to the name of my local machine:

```
PS SQLSERVER:\SQL\ROGUE\DEFAULT> ...
```

Note the starting path in this window. The screen now shows how you could get to the default instance if you were to navigate using the PowerShell console or ISE:

```
PS SQLSERVER:\SQL\<SQL instance name>\DEFAULT>
```

3. Type `dir`. This should give you a list of all objects directly accessible from the current server instance; in our case, from the default SQL Server instance **ROGUE**. Note that `dir` is an alias for the cmdlet `Get-ChildItem`.

```
PS SQLSERVER:\SQL\ROGUE\DEFAULT> dir
Audits
AvailabilityGroups
BackupDevices
Credentials
CryptographicProviders
Databases
Endpoints
JobServer
Languages
LinkedServers
Logins
Mail
ResourceGovernor
Roles
ServerAuditSpecifications
SystemDataTypes
SystemMessages
Triggers
UserDefinedMessages
```

**Downloading the example code**

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.
This is similar to the objects you can find under the instance node in **Object Explorer** in SQL Server Management Studio.

4. While our PowerShell window is open, let's explore the SQL Server PSDrive or the SQL Server data store which PowerShell treats as a series of items. Type `cd \. This will change the path to the root of the current drive, which is our SQL Server PSDrive.

5. Type `dir`. This will list all items accessible from the root SQL Server PSDrive. You should see something similar to the following screen:
6. Close this window.

7. Go back to Management Studio and right-click on one of your user databases.

8. Click on Start PowerShell. Note that this will launch another PowerShell session with a path that points to the database you right-clicked from:

```
PS SQLSERVER:\SQL\ROGUE\DEFAULT> cd \nPS SQLSERVER:\> dir
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Root</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAC</td>
<td>SQLSERVER:\DAC</td>
<td>SQL Server Data-Tier Application Component</td>
</tr>
<tr>
<td>DataCollection</td>
<td>SQLSERVER:\DataCollection</td>
<td>SQL Server Data Collection</td>
</tr>
<tr>
<td>SQLPolicy</td>
<td>SQLSERVER:\SQLPolicy</td>
<td>SQL Server Policy Management</td>
</tr>
<tr>
<td>Utility</td>
<td>SQLSERVER:\Utility</td>
<td>SQL Server Utility</td>
</tr>
<tr>
<td>SQLRegistration</td>
<td>SQLSERVER:\SQLRegistration</td>
<td>SQL Server Registrations</td>
</tr>
<tr>
<td>SQL</td>
<td>SQLSERVER:\SQL</td>
<td>SQL Server Database Engine</td>
</tr>
<tr>
<td>SSIS</td>
<td>SQLSERVER:SSIS</td>
<td>SQL Server Integration Services</td>
</tr>
<tr>
<td>XEvent</td>
<td>SQLSERVER:\XEvent</td>
<td>SQL Server Extended Events</td>
</tr>
<tr>
<td>DatabaseXEvent</td>
<td>SQLSERVER:\DatabaseXEvent</td>
<td>SQL Server Extended Events</td>
</tr>
<tr>
<td>SCLAS</td>
<td>SQLSERVER:\SCLAS</td>
<td>SQL Server Analysis Services</td>
</tr>
</tbody>
</table>

Note the starting path of this window is different from the starting path where you first launched PowerShell in the earlier steps. If you type `dir` from this location, you will see all items that are under the `AdventureWorks2014` database.

```
PS SQLSERVER:\SQL\ROGUE\DEFAULT\Databases\AdventureWorks2014> dir
ApplicationRoles
Assemblies
AsymmetricKeys
Certificates
DatabaseAuditSpecifications
Defaults
ExtendedProperties
ExtendedStoredProcedures
Federations
FileGroups
FullTextCatalogs
FullTextStoplists
LogFiles
PartitionFunctions
PartitionSchemes
PlanGuides
Roles
Rules
Schemas
```
You can see some of the items enumerated in this screenshot in SQL Server Management Studio **Object Explorer**, if you expand the **AdventureWorks2014** database node:

![Object Explorer](image)

**How it works...**

When PowerShell is launched through SSMS, a context-sensitive PowerShell session is created and it automatically loads the SQLPS module. This will be evident in the prompt, which by default shows the current path of the object from which the **Start PowerShell** menu item was clicked from.

```
PS SQLSERVER:\SQL\ROGUE\DEFAULT\Databases\AdventureWorks2014>
```
The SQLPS module was not always loaded when PowerShell was launched from SSMS. With SQL Server 2008/2008 R2 it was shipped with a SQLPS utility, which is also referred to as a mini shell. When you started PowerShell from SSMS, it was not a full PowerShell console that was launched. It was a constrained, closed shell preloaded with SQL Server extensions was loaded. This shell was meant to be used for SQL Server only, which proved to be quite limiting because DBAs and developers often need to load additional snapins and modules in order to integrate SQL Server with other systems through PowerShell. At that time, the alternative way was to launch a full-fledged PowerShell session and depending on your PowerShell version, either load snapins or load the SQLPS module.

Since SQL Server 2012, the original constrained mini shell has been deprecated. When you launch a PowerShell session from SSMS in SQL Server 2012 onwards, what is launched is the full-fledged PowerShell console, with the updated SQLPS module loaded by default.

Once the SQLPS module is loaded, SQL Server becomes exposed as a PowerShell Drive (PSDrive), which allows traversing of objects as if they are folders and files. Familiar commands for traversing directories are supported in this provider, such as dir or ls. Note that these familiar commands are often just aliases to the real cmdlet name, in this case, Get-ChildItem.

When you launch PowerShell from SSMS, you can immediately start navigating the SQL Server PowerShell hierarchy.

**Discovering SQL-related cmdlets and modules**

In order to be good at working with SQL Server and PowerShell, knowing how to explore and discover cmdlets, snapins, and modules is necessary.

**Getting ready**

Launch *PowerShell ISE* as administrator. If you prefer the console, you can also launch that instead, but ensure you are running it as administrator.

**How to do it...**

In this recipe, we will explore SQL-related cmdlets and modules:

1. To find out how many SQL-related cmdlets are in your system, type the following in your PowerShell editor and run:

   #how many commands from modules that
   #have SQL in the name
   (Get-Command -Module "*SQL*" -CommandType Cmdlet).Count
2. To list the SQL-related cmdlets, type the following in your PowerShell editor and run:

```
#list all the SQL-related commands
Get-Command -Module "*SQL*" -CommandType Cmdlet |
Select-Object CommandType, Name, ModuleName |
Sort-Object -Property ModuleName, CommandType, Name |
Format-Table -AutoSize
```

3. To see which of these modules are loaded in your PowerShell session, type the following in your editor and run:

```
Get-Module -Name "*SQL*"
```

If you have already used any of the cmdlets in the previous step, then you should see both SQLPS and SQLASCMDLETS. Otherwise, you will need to load these modules before you can use them.

4. To explicitly load these modules, type the following and run:

```
Import-Module -Name "SQLPS"
```

Note that SQLASCMDLETS will be loaded when you load SQLPS.

## How it works...

At the core of PowerShell, we have cmdlets. A cmdlet (pronounced commandlet) is defined in MSDN as **lightweight command that is used in the Windows PowerShell environment**. It can be a compiled, reusable .NET code or an advanced function, or it can be a workflow that typically performs a very specific task. All cmdlets follow the verb-noun naming notation.

PowerShell ships with many cmdlets. In addition, many applications now also ship with their own cmdlets. For example, SharePoint has a fair number of PowerShell cmdlets that help with installation, configuration, and administration of the farm, sites, and everything in between. A list of cmdlets for SharePoint 2013 can be found at [https://technet.microsoft.com/en-us/library/ff678226.aspx](https://technet.microsoft.com/en-us/library/ff678226.aspx).

A legacy way of extending PowerShell is by registering additional snapins. A **Snapin** is a binary, or a DLL, that can contain a cmdlet. You can create your own snapin by building your own .NET source, compiling, and registering the snapin. You will always need to register snapins before you can use them. Snapins are a popular way of extending PowerShell.

The following table summarizes common tasks with snapins:

<table>
<thead>
<tr>
<th>Task</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>List loaded Snapins</td>
<td>Get-PSSnapin</td>
</tr>
<tr>
<td>List Installed Snapins</td>
<td>Get-PSSnapin -Registered</td>
</tr>
<tr>
<td>Show commands in a Snapin</td>
<td>Get-Command -Module &quot;SnapinName&quot;</td>
</tr>
<tr>
<td>Load a specific Snapin</td>
<td>Add-PSSnapin &quot;SnapinName&quot;</td>
</tr>
</tbody>
</table>
Since PowerShell v2, modules are introduced as the improved and preferred method of extending PowerShell. A module is a package that can contain cmdlets, providers, functions, variables, and aliases. In PowerShell v2, modules are not loaded by default, so required modules need to be explicitly imported.

Common tasks with modules are summarized in the following table:

<table>
<thead>
<tr>
<th>Task</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>List loaded Modules</td>
<td>Get-Module</td>
</tr>
<tr>
<td>List Installed Modules</td>
<td>Get-Module -ListAvailable</td>
</tr>
<tr>
<td>Show commands in a Module</td>
<td>Get-Command -Module &quot;ModuleName&quot;</td>
</tr>
<tr>
<td>Load a specific Module</td>
<td>Import-Module -Name &quot;ModuleName&quot;</td>
</tr>
</tbody>
</table>

One of the improved features of PowerShell v3 onwards is support for autoloading modules. You do not need to always explicitly load modules before using the contained cmdlets. Using the cmdlets in your script is enough to trigger PowerShell to load the module that contains it.

SQL Server 2014 modules are located at `PowerShell | Modules` in the install directory.

<table>
<thead>
<tr>
<th>Name</th>
<th>Date modified</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLASCMDLETS</td>
<td>8/21/2014 7:13 PM</td>
<td>File folder</td>
</tr>
<tr>
<td>SQLPS</td>
<td>8/21/2014 7:13 PM</td>
<td>File folder</td>
</tr>
</tbody>
</table>

**There's more...**

You can get a list of SQLPS and SQLASCMDLETS by running the following command:

```powershell
Get-Command -CommandType Cmdlet -Module SQLPS,SQLASCMDLETS | Select-Object Name, Module | Sort-Object Module, Name | Format-Table -AutoSize
```

Here's the list of cmdlets as of this version of SQL Server 2014:

<table>
<thead>
<tr>
<th>CommandType</th>
<th>Name</th>
<th>ModuleName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cmdlet</td>
<td>Add-RoleMember</td>
<td>SQLASCMDLETS</td>
</tr>
<tr>
<td>Cmdlet</td>
<td>Backup-ASDatabase</td>
<td>SQLASCMDLETS</td>
</tr>
<tr>
<td>Cmdlet</td>
<td>Invoke-ASCmd</td>
<td>SQLASCMDLETS</td>
</tr>
<tr>
<td>Cmdlet</td>
<td>Invoke-ProcessCube</td>
<td>SQLASCMDLETS</td>
</tr>
<tr>
<td>Cmdlet</td>
<td>Invoke-ProcessDimension</td>
<td>SQLASCMDLETS</td>
</tr>
</tbody>
</table>
Getting Started with SQL Server and PowerShell

Cmdlet Invoke-ProcessPartition                SQLASCMDLETS
Cmdlet Merge-Partition                        SQLASCMDLETS
Cmdlet New-RestoreFolder                      SQLASCMDLETS
Cmdlet New-RestoreLocation                    SQLASCMDLETS
Cmdlet Remove-RoleMember                      SQLASCMDLETS
Cmdlet Restore-ASDatabase                      SQLASCMDLETS
Cmdlet Add-SqlAvailabilityDatabase            SQLPS
Cmdlet Add-SqlAvailabilityGroupListenerStaticIp SQLPS
Cmdlet Add-SqlFirewallRule                    SQLPS
Cmdlet Backup-SqlDatabase                     SQLPS
Cmdlet Convert-UrnToPath                      SQLPS
Cmdlet Decode-SqlName                         SQLPS
Cmdlet Disable-SqlAlwaysOn                    SQLPS
Cmdlet Enable-SqlAlwaysOn                     SQLPS
Cmdlet Encode-SqlName                         SQLPS
Cmdlet Get-SqlCredential                      SQLPS
Cmdlet Get-SqlDatabase                         SQLPS
Cmdlet Get-SqlInstance                        SQLPS
Cmdlet Get-SqlSmartAdmin                      SQLPS
Cmdlet Get-SqlSmartAdmin                      SQLPS
Cmdlet Invoke-PolicyEvaluation                 SQLPS
Cmdlet Invoke-Sqlcmd                          SQLPS
Cmdlet Join-SqlAvailabilityGroup              SQLPS
Cmdlet New-SqlAvailabilityGroup               SQLPS
Cmdlet New-SqlAvailabilityGroupListener       SQLPS
Cmdlet New-SqlAvailabilityReplica             SQLPS
Cmdlet New-SqlBackupEncryptionOption          SQLPS
Cmdlet New-SqlCredential                       SQLPS
Cmdlet New-SqlHADREndpoint                    SQLPS
Cmdlet Remove-SqlAvailabilityDatabase         SQLPS
Cmdlet Remove-SqlAvailabilityGroup            SQLPS
Cmdlet Remove-SqlAvailabilityGroupListener    SQLPS
Cmdlet Remove-SqlAvailabilityReplica          SQLPS
Cmdlet Remove-SqlCREDENTIAL                   SQLPS
Cmdlet Remove-SqlFirewallRule                 SQLPS
Cmdlet Restore-SqlDatabase                    SQLPS
Cmdlet Resume-SqlAvailabilityDatabase          SQLPS
Cmdlet Set-SqlAuthenticationMode               SQLPS
Cmdlet Set-SqlAvailabilityGroup               SQLPS
Cmdlet Set-SqlAvailabilityGroupListener       SQLPS
Cmdlet Set-SqlAvailabilityReplica             SQLPS
Cmdlet Set-SqlCredential                       SQLPS
Chapter 1

Cmdlet Set-SqlHADREndpoint                      SQLPS
Cmdlet Set-SqlNetworkConfiguration              SQLPS
Cmdlet Set-SqlSmartAdmin                        SQLPS
Cmdlet Start-SqlInstance                        SQLPS
Cmdlet Stop-SqlInstance                         SQLPS
Cmdlet Suspend-SqlAvailabilityDatabase          SQLPS
Cmdlet Switch-SqlAvailabilityGroup              SQLPS
Cmdlet Test-SqlAvailabilityGroup                SQLPS
Cmdlet Test-SqlAvailabilityReplica              SQLPS
Cmdlet Test-SqlDatabaseReplicaState             SQLPS
Cmdlet Test-SqlSmartAdmin                       SQLPS

To learn more about these cmdlets, use the Get-Help cmdlet. For example, here's the command to learn more about Invoke-Sqlcmd:

Get-Help Invoke-Sqlcmd
Get-Help Invoke-Sqlcmd -Detailed
Get-Help Invoke-Sqlcmd -Examples
Get-Help Invoke-Sqlcmd -Full

You can also check out the MSDN article on SQL Server Database Engine Cmdlets at http://msdn.microsoft.com/en-us/library/cc281847.aspx.

When you load the SQLPS module, several assemblies are loaded into your host.

To get a list of SQLServer-related assemblies loaded with the SQLPS module, use the following script, which will work in both PowerShell v2 and v3:

Import-Module SQLPS –DisableNameChecking

[AppDomain]::CurrentDomain.GetAssemblies() | Where-Object {$_._ FullName -match "SqlServer" } | Select-Object FullName

If you want to run on v3 or newer versions, you can take advantage of the simplified syntax:

Import-Module SQLPS –DisableNameChecking

[AppDomain]::CurrentDomain.GetAssemblies() | Where-Object FullName -Match "SqlServer" | Select-Object FullName
Getting Started with SQL Server and PowerShell

This will show you all the loaded assemblies, including their public key tokens:

<table>
<thead>
<tr>
<th>FullName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft.SqlServer.Smo, Version=12.0.0.0, Culture=neutral, PublicKeyToken=80845dcdd8080cc91</td>
</tr>
<tr>
<td>Microsoft.SqlServer.Omf, Version=12.0.0.0, Culture=neutral, PublicKeyToken=80845dcdd8080cc91</td>
</tr>
<tr>
<td>Microsoft.SqlServer.Management, Version=12.0.0.0, Culture=neutral, PublicKeyToken=80845dcdd8080cc91</td>
</tr>
<tr>
<td>Microsoft.SqlServer.ConnectionInfo, Version=12.0.0.0, Culture=neutral, PublicKeyToken=80845dcdd8080cc91</td>
</tr>
<tr>
<td>Microsoft.SqlServer.SmoExtended, Version=12.0.0.0, Culture=neutral, PublicKeyToken=80845dcdd8080cc91</td>
</tr>
<tr>
<td>Microsoft.SqlServer.Management.RegisteredServers, Version=12.0.0.0, Culture=neutral, PublicKeyToken=80845dcdd8080cc91</td>
</tr>
<tr>
<td>Microsoft.SqlServer.Management.Smo, Version=12.0.0.0, Culture=neutral, PublicKeyToken=80845dcdd8080cc91</td>
</tr>
<tr>
<td>Microsoft.SqlServer.ConnectionInfo, Version=12.0.0.0, Culture=neutral, PublicKeyToken=80845dcdd8080cc91</td>
</tr>
<tr>
<td>Microsoft.SqlServer.SmoExtended, Version=12.0.0.0, Culture=neutral, PublicKeyToken=80845dcdd8080cc91</td>
</tr>
<tr>
<td>Microsoft.SqlServer.Management.RegisteredServers, Version=12.0.0.0, Culture=neutral, PublicKeyToken=80845dcdd8080cc91</td>
</tr>
</tbody>
</table>

Creating a SQL Server Instance Object

Most of what you will need to do in SQL Server will require a connection to an instance. One way to do this is by creating an instance object via SMO.

Getting ready

Open up your PowerShell console, **PowerShell ISE**, or your favorite PowerShell editor.

You will need to note what your instance name is. If you have a default instance, you can use your machine name. If you have a named instance, the format will be `<machine name>`\`<instance name>`.

How to do it...

If you are connecting to your instance using Windows authentication and using your current Windows login, the steps are as follows:

1. Import the SQLPS module:
   ```powershell
   #import SQLPS module
   Import-Module SQLPS -DisableNameChecking
   $VerbosePreference = "SilentlyContinue"
   ```
2. Store your instance name in a variable:

```powershell
# create a variable for your instance name
$instanceName = "localhost"
```

3. If you are connecting to your instance using Windows authentication from the account you are logged in as:

```powershell
# create your server instance
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.Server -ArgumentList $instanceName
```

If you are connecting using **SQL Authentication**, you will need to know the username and password that you will use to authenticate. In this case, you will need to add the following code, which will set the connection to mixed mode and prompt for the username and password:

```powershell
# set connection to mixed mode
$server.ConnectionContext.set_LoginSecure($false)

# set the login name
# of course we don't want to hardcode credentials here
# so we will prompt the user
# note password is passed as a SecureString type
$credentials = Get-Credential

# remove leading backslash in username
$login = $credentials.UserName $server.ConnectionContext.set_Login($login)
$server.ConnectionContext.set_SecurePassword($credentials.Password)

# check connection string
# note though that this outputs your password in clear text
$server.ConnectionContext.ConnectionString

Write-Verbose "Connected to $($server.Name)"
Write-Verbose "Logged in as $($server.ConnectionContext.Login)"
```

**How it works...**

Before you can access or manipulate SQL Server programmatically, you will often need to create references to its objects. At the most basic level is the server.
The server instance is using the type `Microsoft.SqlServer.Management.Smo.Server`. By default, connections to the server are using a trusted connection, meaning it uses the Windows account you're currently using when you log into the server. So all it needs is the instance name in its argument list:

```powershell
# create your server instance
$server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.Server -ArgumentList $instanceName
```

However, if you need to connect using a SQL login, you will need to set the `ConnectionContext.LoginSecure` property of the SMO Server class setting to `false`:

```powershell
# set connection to mixed mode
# note that this authentication will fail if mixed mode
# is not enabled in SQL Server
$server.ConnectionContext.set_LoginSecure($false)
```

You will also need to explicitly set the username and the password. The best way to accomplish this is to prompt the user for the credentials:

```powershell
# prompt
$credentials = Get-Credential
```

The `Get-Credential` cmdlet will display a popup window that will capture the login and password entered by the user:

```powershell
$login = $credentials.UserName
```

Once we have the login, we can pass it to the `set_Login` method. The password is already a SecureString type, which means it is encrypted. This is the data type required by the `set_SecurePassword` method, so no further conversion is needed. The commands are as follows:

```powershell
$server.ConnectionContext.set_Login($login)
$server.ConnectionContext.set_SecurePassword($credentials.Password)
```

Should you want to hardcode the username and just prompt for the password, you can also do this:

```powershell
$login = "belle"
# prompt
$credentials = Get-Credential -Credential $login
```

In the script, you will also notice that we are using `Write-Verbose` instead of `Write-Host` to display our results. This is because we want to control the output without always going back to our script and removing the `Write-Host` command.
By default, the script will not display any output, that is, the $VerbosePreference special variable is set to SilentlyContinue. If you want to run the script in verbose mode, you simply need to add this in the beginning of your script:

$VerbosePreference = "Continue"

When you are done, you just need to change the value to SilentlyContinue:

$VerbosePreference = "SilentlyContinue"

See also

- The recipe Loading SMO assemblies.
- The recipe Creating a SQL Server Instance Object.

Exploring SMO Server Objects

SMO comes with a hierarchy of objects that are accessible programmatically. For example, when we create an SMO server variable, we can then access databases, logins, and database level triggers. Once we get a handle of individual databases, we can then traverse the tables, stored procedures and views that it contains. Since many tasks involve SMO objects, you will be at an advantage if you know how to discover and navigate these objects.

Getting ready

Open up your PowerShell console, PowerShell ISE, or your favorite PowerShell editor.

You will also need to note what your instance name is. If you have a default instance, you can use your machine name. If you have a named instance, the format will be $machine name\$instance name$.

How to do it...

In this recipe, we will start exploring the hierarchy of objects with SMO:

1. Import the SQLPS module as follows:
   
   ```powershell
   Import-Module SQLPS -DisableNameChecking
   ```

2. Create a server instance as follows:
   
   ```powershell
   $instanceName = "localhost"
   #code below all in one line
   $server = New-Object -TypeName Microsoft.SqlServer.Management.Smo.Server -ArgumentList $instanceName
   ```
3. Get the SMO objects directly accessible from the $server object:

```powershell
$server | Get-Member -MemberType "Property" | Where-Object Definition -Like "^Smo*"
```

If you are using PowerShell v2, you will have to change the `Where-Object` cmdlet usage to use the curly braces `{}` and the `$_` variable:

```powershell
Where-Object {$_.Definition -like "^Smo*"}
```

4. Now, let's check SMO objects under databases:

```powershell
$server.Databases | Get-Member -MemberType "Property" | Where-Object Definition -Like "^Smo*"
```

5. To check out the tables, you can type and execute the following:

```powershell
$server.Databases["AdventureWorks2014"].Tables | Get-Member -MemberType "Property" | Where-Object Definition -Like "^Smo*"
```

**How it works...**

SMO contains a hierarchy of objects. At the very top there is a server object, which in turn contains objects such as Databases, Configuration, SqlMail, LoginCollection, and so on. These objects in turn contain other objects, for example, Databases is a collection that contains Database objects, and a Database contains Tables.

You can check out the SMO Object Model Diagram from the MSDN at https://msdn.microsoft.com/en-ca/library/ms162209.aspx.

One way to navigate through the hierarchy is by creating a server instance first. From here, you can use Get-Member to figure out which properties belong to that object. Once you find out, you can start creating additional variables for the member objects and then use Get-Member on them. Lather, rinse, and repeat.

**See also**

- The recipe *Loading SMO assemblies*.
- The recipe *Creating a SQL Server Instance Object*.
Where to buy this book

You can buy SQL Server 2014 with Powershell v5 Cookbook from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.